Swifter - 100 个 Swift 必备 tips

Table of Contents

介绍

> 虽然我们都希望能尽快开始在 Swift 的世界里遨游， 但是我觉得仍然有必要花一些时间将本书的写作目的和 适合哪些读者进行必要说明。我不喜欢自吹自擂，也无法 承担“骗子”的骂名。在知识这件严肃的事情上，我并不希 望对读者产生任何的误导。作为读者，您一定想要找的是一本 合适自己的书；而作为作者，我也希望找到自己的伯乐和子期。

为什么要写这本书

中文的科技书籍太少了，内容也太浅了。这是国内市场尴尬的现状：真正有技术的大牛不在少数，但他们很多并不太愿意通过出书的方式来分享他们的知识：一方面是回报率实在太低，另一方面是出版的流程过于繁琐。这就导致了市面上充斥了一些习惯于出版业务，但是却丝毫无视质量和素质的流氓作者和图书。

特别是对于 Swift 语言来说，这个问题尤其严重。iOS 开发不可谓不火热，每天都有大量的开发者涌入这个平台。而 Swift 的发布更使得原本高温的市场更上一层楼。但是市面上随处可见的都是各种《开发指南》《权威指南》或者《21天学会XXX》系列的中文资料。这些图书大致都是对官方文档的翻译，并没有什么实质的见解，可以说内容单一，索然无味。作为读者，很难理解作者写作的重心和目的 (其实说实话，大部分情况下这类书的作者自己都不知道写作的重心和目的是什么)，这样的“为了出版而出版”的图书可以说除了增加世界的熵以外，几乎毫无价值。

如果想要入门 Swift 语言，阅读 Apple 官方教程和文档无论从条理性和权威性来说，都是更好的选择。而中国的 Cocoa 开发者社区也以令人惊叹的速度完成了对文档的高品质翻译，这在其他任何国家都是让人眼红的一件事情。因此，如果您是初学程序设计或者 Swift 语言，相比起那些泯灭良心 (抱歉我用了这个词，希望大家不要对号入座) 的“入门书籍”，我更推荐您看这份翻译后的官方文档，这是非常珍惜和宝贵的资源。

说到这里，可以谈谈这本《Swifter - 100 个 Swift 必备 tips》的写作目的

了。很多 Swift 的学习者 – 包括新接触 Cocoa/Cocoa Touch 开发的朋友，以及之前就使用 Objective-C 的朋友 – 所共同面临的一个问题是，入门以后应该如何进一步提高。也许你也有过这样的感受：在阅读完 Apple 的教程后，觉得自己已经学会了 Swift 的语法和使用方式，你满怀信心地打开 Xcode，新建了一个 Swift 项目，想写点什么，却发现实际上满不是那么回事。你需要联想 Optional 应该在什么时候使用，你可能发现本已熟知 API 突然不太确定要怎么表达，你可能遇到怎么也编译不了的问题但却不知如何改正。这些现象都非常正常，因为教程是为了展示某个语法点而写的，而几乎不涉及实际项目中应该如何使用的范例。本书的目的就是为广大已经入门了 Swift 的开发者提供一些参考，以期能迅速提升他们在实践中的能力。因为这部分的中级内容是我自己力所能及，有自信心能写好的；也是现在广大 Swift 学习者所急缺和需要的。

这本书是什么

本书是 Swift 语言的知识点的集合。我自己是赴美参加了 Apple 的 WWDC 14 的，也正是在这届开发者大会上，Swift 横空出世。毫不夸张地说，从 Swift 正式诞生的第一分钟开始，我就在学习这门语言。虽然天资驽钝，不得其所，但是在这段集中学习和实践的时间里，也还算总结了一些心得，而我把这些总结加以整理和示例，以一个个的小技巧和知识点的形式，编写成了这本书。全书共有 100 节，每一节都是一个相对独立的主题，涵盖了一个中高级开发人员需要知道的 Swift 语言的方方面面。

这本书非常适合用作官方文档的参考和补充，也会是中级开发人员很喜爱的 Swift 进阶读本。具体每个章节的内容，可以参看本书的目录。

这本书不是什么

这本书**不是** Swift 的入门教程，也**不会**通过具体的完整实例引导你用 Swift 开发出一个像是计算器或者记事本这样的 app。这本书的目的十分纯粹，就是探索那些不太被人注意，但是又在每天的开发中可能经常用到的 Swift 特性的。这本书并不会系统地介绍 Swift 的语法和特性，因为基于本书的写作目的和内容特点，采用松散的模式和非线性的组织方式会更加适合。

换言之，如果你是想找一本 Swift 从零开始的书籍，那这本书不应该是你的选择。你可以在阅读 Apple 文档后再考虑回来看这本书。

组织形式和推荐的阅读方式

100 个 tips 其实不是一个小数目。本书的每个章节的内容是相对独立的，也就是说你没有必要从头开始看，随手翻开到任何一节都是没问题的。当然，按顺序看是最理想的阅读方式，因为在写作时我特别注意了让靠前的章节不涉及后面章节的内容；另一方面，位置靠后的章节如果涉及到之前章节

内容的话，我添加了跳转到相关章节的链接，这可以帮助迅速复习和回顾之前的内容。我始终坚信不断的重复和巩固，是真正掌握知识的唯一途径。

本书的电子版的目录是可以点击跳转的，您可以通过目录快速地在不同章节之间导航。如果遇到您不感兴趣或者已经熟知的章节，您也完全可以暂时先跳过去，这不会影响您对本书的阅读和理解。

建议您一边阅读本书时一边开启 Xcode 环境并且对每一章节中的代码进行验证，这有利于您真正理解代码示例想表达的意思，也有利于记忆的形成。每一段代码示例都不太长，但却是经过精心准备，能很好地说明章节内容的，希望您能在每一章里都能通过代码和我进行心灵上的“对话”。

排版和字体

因为受限于电子书出版平台，因此本书的 PDF 版本在字体和行距的控制上我很不满意。虽然和出版平台进行过一些交涉，但是似乎距离添加必要的控制选项还会有一段时间。我一度考虑过不使用 PDF 来发布这本书，但是有读者朋友反应还是想要一个 PDF 版，因此最后决定保留。但是个人推荐的方式还是使用 epub 或者 mobi 格式的电子书的方式来阅读本书，这样您可以自由地控制本书的字体和格式，会舒服一些。

本书提供了试看的章节，您应该可以在本书的购买页面上看到试看章节的下载链接。您可以通过试看确认本书在您的设备上的表现。

代码运行环境

书中每一章基本都配有代码示例的说明。这些代码一般来说包括 Objective-C 或者 Swift 的代码。理论上来说所有代码都可以在 Swift 1.2 (也就是 Xcode 6.3) 版本环境下运行。当然 因为 Swift 版本变化很快，可能部分代码需要微调或者结合一定的上下文环境才能运行，但我相信这种调整是显而易见的。如果您发现明显的代码错误和无法运行的情况，欢迎到本书的 issue 页面 上提出，我将尽快修正。

如果没有特别说明，这些代码在 Playground 和项目中都应该可以运行，并拥有同样表现的。但是也存在一些代码只能在 Playground 或者项目文件中才能正确工作的情况，这主要是因为平台限制的因素，如果出现这种情况，我都会在相关章节中特别加以说明。

勘误和反馈

Swift 仍然在高速发展和不断变化中，本书最早版本基于 Swift 1.0，当前版本基于 Swift 1.2。随着 Swift 的新特性引入以及错误修正，本书难免会存在部分错误，其中包括为对应的更新纰漏或者部分内容过时的情况。虽然我会

随着 Swift 的发展继续不断完善和修正这本书，但是这个过程亦需要时间，请您谅解。

另外由于作者水平有限，书中也难免会出现一些错误，如果您在阅读时发现了任何问题，可以到这本书 issue 页面进行反馈。我将尽快确认和修正。得益于电子书的优势，本书的读者都可以在本书更新时免费获得所有的新内容。每次更新的变更内容将会写在本书的更新一节中，您也可以在更新内容页面上找到同样的列表。

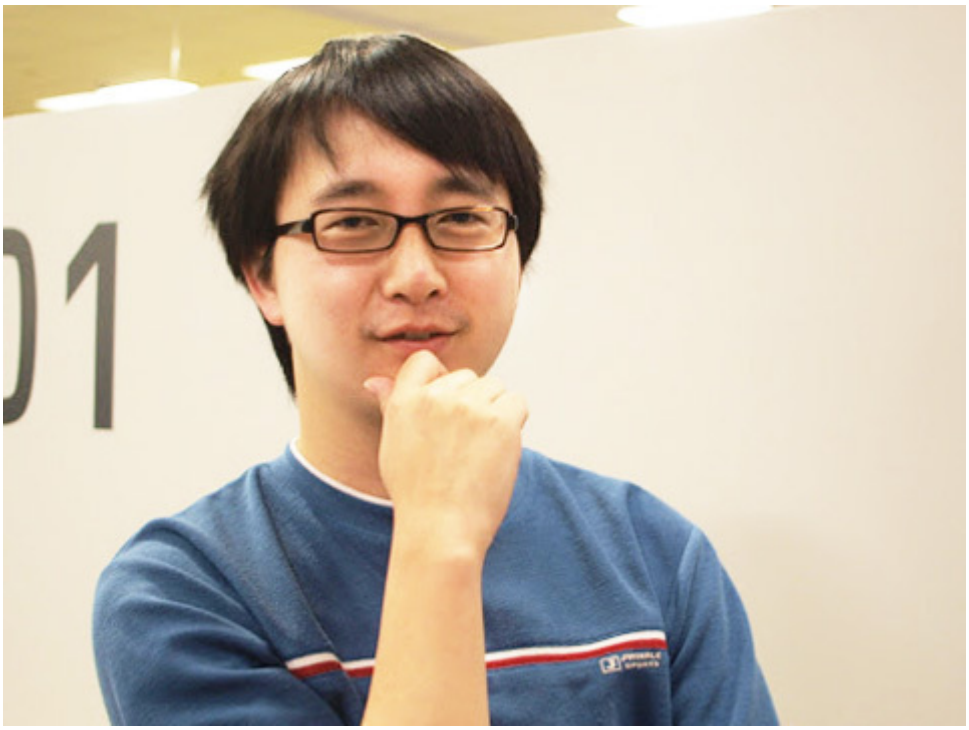
版权问题

首先想感谢您购买了这本书。在国内知识产权保护不足的现状下，我自知出版这样一本没有任何保护措施的电子书可能无异于飞蛾扑火。我其实是怀着忐忑的心情写下这些文字的，小心翼翼地希望没有触动到太多人。如果您不是通过 Gumroad，Leanpub 或者是 SelfStore 购买，而拿到这本书的话，您应该是盗版图书的受害者。这本书所提供的知识和之后的服务我想应该是超过它的售价 (大约是一杯星巴克咖啡的价格) 的，在阅读前还请您再三考虑。

另外，这本书暂时只有电子版。如果合适，我会考虑联系国内出版社进行纸质版的出版，但那应该是另外一个故事了。

最后一部分是我的个人简介，您可以跳过不看，而直接开始本书的第一章。

作者简介



王巍 (onevcat) 是来自中国的一线 iOS 开发者，毕业于清华大学。在校期间

就开始进行 iOS 开发，拥有丰富的 Cocoa 和 Objective-C 开发经验，另外他也活跃于使用 C# 的 Unity3D 游戏开发界。曾经开发了《小熊推金币》，《Pomo Do》等一系列优秀的 iOS 游戏和应用。在业余时间，王巍会在 OneV's Den 撰写博客，分享他在开发中的一些心得和体会。另外，王巍还是翻译项目 objc 中国 的组织者和管理者，为中国的 Objective-C 社区的发展做出了贡献。同时，他也很喜欢为开源社区贡献代码，是著名的 Xcode 插件 VVDocumenter 的作者。

现在王巍旅居日本，并就职于即时通讯软件公司 Line，从事 iOS 开发工作，致力于为全世界带来更好体验和功能的应用。如果您需要进一步了解作者的话，可以访问他的资料页面。

Selector

@selector 是 Objective-C 时代的一个关键字，它可以将一个方法转换并赋值给一个 SEL 类型，它的表现很类似一个动态的函数指针。在 Objective-C 时 selector 非常常用，从设定 target-action，到自举询问是否响应某个方法，再到指定接受通知时需要调用的方法等等，都是由 selector 来负责的。在 Objective-C 里生成一个 selector 的方法一般是这个样子的：

```
-(void) callMe {
    //...
}

-(void) callMeWithParam:(id)obj {
    //...
}

SEL someMethod = @selector(callMe);
SEL anotherMethod = @selector(callMeWithParam:);

// 或者也可以使用 NSSelectorFromString
// SEL someMethod = NSSelectorFromString(@"callMe");
// SEL anotherMethod = NSSelectorFromString(@"callMeWithParam:");
```

一般为了方便，很多人会选择使用 @selector，但是如果要追求灵活的话，可能会更愿意使用 NSSelectorFromString 的版本 – 因为我们可以在运行时动态生成字符串，从而通过方法的名字来调用到对应的方法。

在 Swift 中没有 @selector 了，我们要生成一个 selector 的话现在只能使用字符串。Swift 里对应原来 SEL 的类型是一个叫做 Selector 的结构体，它提供了一个接受字符串的初始化方法。像上面的两个例子在 Swift 中等效的写法是：

```
func callMe() {
```

```
    //...
}

func callMeWithParam(obj: AnyObject!) {
    //...
}

let someMethod = Selector("callMe")
let anotherMethod = Selector("callMeWithParam:")
```

和 Objective-C 时一样，记得在 callMeWithParam 后面加上冒号 (:)，这才是完整的方法名字。多个参数的方法名也和原来类似，是这个样子：

```
func turnByAngle(theAngle: Int, speed: Float) {
    //...
}

let method = Selector("turnByAngle:speed:")
```

另外，因为 Selector 类型实现了 StringLiteralConvertible，因此我们甚至可以不使用它的初始化方法，而直接用一个字符串进行赋值，就可以完成创建了。

最后需要注意的是，selector 其实是 Objective-C runtime 的概念，如果你的 selector 对应的方法只在 Swift 中可见的话 (也就是说它是一个 Swift 中的 private 方法)，在调用这个 selector 时你会遇到一个 unrecognized selector 错误：

这是错误代码

```
private func callMe() {
    //...
}

NSTimer.scheduledTimerWithTimeInterval(1, target: self,
            selector:"callMe", userInfo: nil, repeats: true)
```

正确的做法是在 private 前面加上 @objc 关键字，这样运行时就能找到对应的方法了。

```
@objc private func callMe() {
    //...
}

NSTimer.scheduledTimerWithTimeInterval(1, target: self,
            selector:"callMe", userInfo: nil, repeats: true)
```

另外，如果方法的第一个参数有外部变量的话，在通过字符串生成 Selector

时还有一个约定，那就是在方法名和第一个外部参数之间加上 with：

```
func aMethod(external paramName: AnyObject!) { ... }
```

想获取对应的获取 Selector，应该这么写：

```
let s = Selector("aMethodWithExternal:")
```

Sequence

Swift 的 for...in 可以用在所有实现了 SequenceType 的类型上，而为了实现 SequenceType 你首先需要实现一个 GeneratorType。比如一个实现了反向的 generator 和 sequence 可以这么写：

```
// 先定义一个实现了 GeneratorType protocol 的类型
// GeneratorType 需要指定一个 typealias Element
// 以及提供一个返回 Element? 的方法 next()
class ReverseGenerator: GeneratorType {
    typealias Element = Int

    var counter: Element
    init<T>(array: [T]) {
        self.counter = array.count - 1
    }

    init(start: Int) {
        self.counter = start
    }

    func next() -> Element? {
        return self.counter < 0 ? nil : counter--
    }
}

// 然后我们来定义 SequenceType
// 和 GeneratorType 很类似，不过换成指定一个 typealias Generator
// 以及提供一个返回 Generator? 的方法 generate()
struct ReverseSequence<T>: SequenceType {
    var array: [T]

    init (array: [T]) {
        self.array = array
    }

    typealias Generator = ReverseGenerator
    func generate() -> Generator {
        return ReverseGenerator(array: array)
    }
}

let arr = [0,1,2,3,4]

// 对 SequenceType 可以使用 for...in 来循环访问
for i in ReverseSequence(array: arr) {
    println("Index \(i) is \(arr[i])")
```

```
}
```

输出为

```
Index 4 is 4
Index 3 is 3
Index 2 is 2
Index 1 is 1
Index 0 is 0
```

如果我们想要深究 for...in 这样的方法到底做了什么的话，如果我们将其展开，大概会是下面这个样子：

```
var g = array.generate()
while let obj = g.next() {
    println(obj)
}
```

顺便你可以免费得到的收益是你可以使用像 map , filter 和 reduce 这些方法，因为它们都有对应 SequenceType 的版本：

```
func map<S : SequenceType, T>(source: S,
      transform: (S.Generator.Element) -> T) -> [T]

func filter<S : SequenceType>(source: S,
      includeElement: (S.Generator.Element) -> Bool) -> [S

func reduce<S : SequenceType, U>(sequence: S,
      initial: U, combine: (U, S.Generator.Element) -> U)
```

@autoclosure 和 ??

Apple 为了推广和介绍 Swift，破天荒地为这门语言开设了一个博客(当然我觉着是因为 Swift 坑太多需要一个地方来集中解释)。其中有一篇提到了一个叫做 @autoclosure 的关键词。

@autoclosure 可以说是 Apple 的一个非常神奇的创造，因为这更多地是像在“hack” 这门语言。简单说，@autoclosure 做的事情就是把一句表达式自动地封装成一个闭包 (closure)。这样有时候在语法上看起来就会非常漂亮。

比如我们有一个方法接受一个闭包，当闭包执行的结果为 true 的时候进行打印：

```
func logIfTrue(predicate: () -> Bool) {
    if predicate() {
        println("True")
    }
}
```

在调用的时候，我们需要写这样的代码

```
logIfTrue({return 2 > 1})
```

当然，在 Swift 中对闭包的用法可以进行一些简化，在这种情况下我们可以省略掉 return，写成：

```
logIfTrue({2 > 1})
```

还可以更近一步，因为这个闭包是最后一个参数，所以可以使用尾随闭包 (trailing closure) 的方式把大括号拿出来，然后省略括号，变成：

```
logIfTrue{2 > 1}
```

但是不管那种方式，要么是书写起来十分麻烦，要么是表达上不太清晰，看起来都让人生气。于是 @autoclosure 登场了。我们可以改换方法参数，在参数名前面加上 @autoclosure 关键字：

```
func logIfTrue(@autoclosure predicate: () -> Bool) {
    if predicate() {
        println("True")
    }
}
```

这时候我们就可以直接写：

```
logIfTrue(2 > 1)
```

来进行调用了，Swift 将会吧 2 > 1 这个表达式自动转换为 () -> Bool。这样我们就得到了一个写法简单，表意清楚的式子。

在 Swift 中，有一个非常有用的操作符，可以用来快速地对 nil 进行条件判断，那就是 ??。这个操作符可以判断输入并在当左侧的值是非 nil 的 Optional 值时返回其 value，当左侧是 nil 时返回右侧的值，比如：

```
var level : Int?
var startLevel = 1
```

```
var currentLevel = level ?? startLevel
```

在这个例子中我们没有设置过 level，因此最后 startLevel 被赋值给了 currentLevel。如果我们充满好奇心地点进 ?? 的定义，可以看到 ?? 有两种版本：

```
func ??<T>(optional: T?, @autoclosure defaultValue: () -> T?) -> T?
func ??<T>(optional: T?, @autoclosure defaultValue: () -> T) -> T
```

在这里我们的输入满足的是后者，虽然表面上看 startLevel 只是一个 Int，但是其实在使用时它被自动封装成了一个 () -> Int，有了这个提示，我们不妨来猜测一下 ?? 的实现吧：

```
func ??<T>(optional: T?, @autoclosure defaultValue: () -> T) -> T {
    switch optional {
        case .Some(let value):
            return value
        case .None:
            return defaultValue()
        }
}
```

可能你会有疑问，为什么这里要使用 autoclosure，直接接受 T 作为参数并返回不行么？这正是 autoclosure 的一个最值得称赞的地方。如果我们直接使用 T，那么就意味着在 ?? 操作符真正取值之前，我们就必须准备好一个默认值，这个默认值的准备和计算是会消耗性能的。但是其实要是 optional 不是 nil 的话，我们是完全不需要这个默认值，而会直接返回 optional 解包后的值。这样一来，默认值就白白准备了，这样的开销是完全可以避免的，方法就是将默认值的计算推迟到 optional 判定为 nil 之后。

就这样，我们可以巧妙地绕过条件判断和强制转换，以很优雅的写法处理对 Optional 及默认值的取值了。最后要提一句的是，@autoclosure 并不支持带有输入参数的写法，也就是说只有形如 () -> T 的参数才能使用这个特性进行简化。另外因为调用者往往很容易忽视 @autoclosure 这个特性，所以在写接受 @autoclosure 的方法时还请特别小心，如果在容易产生歧义或者误解的时候，还是使用完整的闭包写法会比较好。

> 在 Swift 1.2 中，@autoclosure 的位置发生了变化。现在 @autoclosure 需要像本文中这样，写在参数名的前面作为参数修饰，而不是在类型前面作为类型修饰。但是现在标准库中的方法签名还是写在了接受的类型前面，这应该是标准库中的疏漏。在我们自己实现一个 autoclosure 时，在类型前修饰的写法在

Swift 1.2 中已经无法编译了。

练习

在 Swift 中，其实 && 和 || 这两个操作符里也用到了 @autoclosure。作为练习，不妨打开 Playground，试试看怎么实现这两个操作符吧?

Optional Chaining

使用 Optional Chaining 可以让我们摆脱很多不必要的判断和取值，但是在使用的时候需要小心陷阱。

因为 Optional Chaining 是随时都可能提前返回 nil 的，所以使用 Optional Chaining 所得到的东西其实都是 Optional 的。比如有下面的一段代码：

```
class Toy {
    let name: String
    init(name: String) {
        self.name = name
    }
}

class Pet {
    var toy: Toy?
}

class Child {
    var pet: Pet?
}
```

在实际使用中，我们想要知道小明的宠物的玩具的名字的时候，可以通过下面的 Optional Chaining 拿到：

```
let toyName = xiaoming.pet?.toy?.name
```

注意虽然我们最后访问的是 name，并且在 Toy 的定义中 name 是被定义为一个确定的 String 而非 String? 的，但是我们拿到的 toyName 其实还是一个 String? 的类型。这是由于在 Optional Chaining 中我们在任意一个 ?. 的时候都可能遇到 nil 而提前返回，这个时候当然就只能拿到 nil 了。

在实际的使用中，我们大多数情况下可能更希望使用 Optional Binding 来直接取值的这样的代码：

```
if let toyName = xiaoming.pet?.toy?.name {
    // 太好了，小明既有宠物，而且宠物还正好有个玩具
}
```

可能单独拿出来看会很清楚，但是只要稍微和其他特性结合一下，事情就会变得复杂起来。来看看下面的例子：

```
extension Toy {
    func play() {
        //...
    }
}
```

我们为 Toy 定义了一个扩展，以及一个玩玩具的 play() 方法。还是拿小明举例子，要是有玩具的话，就玩之：

```
xiaoming.pet?.toy?.play()
```

除了小明也许我们还有小红小李小张等等..在这种时候我们会想要把这一串调用抽象出来，做一个闭包方便使用。传入一个 Child 对象，如果小朋友有宠物并且宠物有玩具的话，就去玩。于是很可能你会写出这样的代码：

这是错误代码

```
let playClosure = {(child: Child) -> () in child.pet?.toy?.pla
```

你会发现这么表意清晰的代码居然无法编译！

问题在于对于 play() 的调用上。定义的时候我们没有写 play() 的返回，这表示这个方法返回 Void (或者写作一对小括号 ()，它们是等价的)。但是正如上所说，经过 Optional Chaining 以后我们得到的是一个 Optional 的结果。也就是说，我们最后得到的应该是这样一个 closure：

```
let playClosure = {(child: Child) -> ()? in child.pet?.toy?.play()}
```

这样调用的返回将是一个 ()? (或者写成 Void? 会更清楚一些)，虽然看起来挺奇怪的，但这就是事实。使用的时候我们可以通过 Optional Binding 来判定方法是否调用成功：

```
if let result: () = playClosure(xiaoming) {
    println("好开心~")
} else {
```

```
    println("没有玩具可以玩 :(")
}
```

func 的参数修饰

在声明一个 Swift 的方法的时候，我们一般不去指定参数前面的修饰符，而是直接声明参数：

```
func incrementor(variable: Int) -> Int {
    return variable + 1
}
```

这个方法接受一个 Int 的输入，然后通过将这个输入加 1，返回一个新的比输入大 1 的 Int。嘛，就是一个简单的 **+1器**。

有些同学在大学的 C 程序设计里可能学过像 ++ 这样的“自增”运算符，再加上做了不少关于“判断一个数被各种前置 ++ 和后置 ++ 折磨后的输出是什么”的考试题，所以之后写代码时也会不自觉地喜欢带上这种风格。于是同样的功能可能会写出类似这样的方法：

这是错误代码

```
func incrementor(variable: Int) -> Int {
    return ++variable
}
```

残念..编译错误。为什么在 Swift 里这样都不行呢？答案是因为 Swift 其实是一门讨厌变化的语言。所有有可能的地方，都被默认认为是不可变的，也就是用 let 进行声明的。这样不仅可以确保安全，也能在编译器的性能优化上更有作为。在方法的参数上也是如此，我们不写修饰符的话，默认情况下所有参数都是 let 的，上面的代码等效为：

```
func incrementor(let variable: Int) -> Int {
    return ++variable
}
```

let 的参数，不能重新赋值这是理所当然的。要让这个方法正确编译，我们需要做的改动是将 let 改为 var：

```
func incrementor(var variable: Int) -> Int {
    return ++variable
}
```

现在我们的 **+1器** 又可以正确工作了：

```
var luckyNumber = 7
let newNumber = incrementor(luckyNumber)
// newNumber = 8

println(luckyNumber)
// luckyNumber 还是 7
```

正如上面的例子，我们将参数写作 var 后，通过调用返回的值是正确的，而 luckyNumber 还是保持了原来的值。这说明 var 只是在方法内部作用，而不直接影响输入的值。有些时候我们会希望在方法内部**直接**修改输入的值，这时候我们可以使用 inout 来对参数进行修饰：

```
func incrementor(inout variable: Int) {
    ++variable
}
```

因为在函数内部就更改了值，所以也不需要返回了。调用也要改变为相应的形式，在前面加上 & 符号：

```
var luckyNumber = 7
incrementor(&luckyNumber)

println(luckyNumber)
// luckyNumber = 8
```

最后，要注意的是参数的修饰是具有传递限制的，就是说对于跨越层级的调用，我们需要保证同一参数的修饰是统一的。举个例子，比如我们想扩展一下上面的方法，实现一个可以累加任意数字的 **+N器** 的话，可以写成这样：

```
func makeIncrementor(addNumber: Int) -> ((inout Int) -> ()) {
    func incrementor(inout variable: Int) -> () {
        variable += addNumber;
    }
    return incrementor;
}
```

外层的 makeIncrementor 的返回里也需要在参数的类型前面明确指出修饰词，以符合内部的定义，否则将无法编译通过。

About Leanpub

What Is Leanpub?
Blog
Team
Buzz
Testimonials
Podcast
Press
Contact Us

Readers

In-Progress & Serial Publishing
Ebook Formats With No DRM
Variable Pricing
100% Happiness Guarantee
Interacting With Authors
The Leanpub App
Kindle Support
Reader FAQ
Reader Help

Authors

Leanpub For Authors
'How To' Introduction
Writing With Leanpub
Writing In Word
Publishing With Leanpub
Royalties and Payments
Leanpub Book Pricing
Uploading A Book
Packaging Books With Videos
The Lean Publishing Manifesto
Author Manual
Author FAQ
Author Help

Books

Agile
Data Science
Computer Programming
Fiction
Non-Fiction
More...

More

Leanpub for Causes
Publishers